Linux forward driver

1.7.16

2026-06-30

Introduction

Support for FDxxx cards in Linux is provided using the open-source **forward** driver.

The driver supports the following types of boards:

- FD722 - 2x inputs, 2x outputs, 3G-SDI
- FD722M2 - 1x input, 1x output, 1x bidirectional (configurable) input/output, 3G-SDI
- FD722BP - 2x inputs, 2x outputs with hardware signal bypass
- FD788 - 8x configurable 3G-SDI inputs/outputs
- FD922 - 2x inputs, 2x outputs, 12G-SDI
- FD722 - 2x HDMI 2.0 inputs
- FD940 - 4x HDMI 2.0 inputs
- FD2110 - 2x configurable 12G-SDI inputs/outputs, 2x 25G-Ethernet interfaces

The driver consists of the following components:

- The `forward` kernel module provides the basic functionality of the boards - I/O configuration, firmware loading, etc.
- The `forward-v4l2` kernel module - provides video processing via the Video4Linux2 API - can be used both by SoftLab-NSK and third-party software (ffmpeg, gstreamer, OBS, etc.).
- The `forward-alsa` kernel module - provides audio processing via the ALSA API - can be used by third-party software as a sound card (pulseaudio, pipewire, gstreamer, OBS, etc.).
- The `forward-ethernet` kernel module - provides a system ethernet adapter for the FD2110 card.
- The `forward-flash` application for updating the devices firmware

The kernel modules are built from source code during installation using the DKMS subsystem. This subsystem allows you to automatically build a driver for all Linux kernels that are installed on the system, and also automatically rebuilds the driver during kernel updates.

Minimal system requirements

- amd64 architecture (x86_64):
- linux 5.4
- glibc 2.28
- dkms 2.8

Driver installation

The driver provided in three versions:

- DEB packages for debian-based systems (Debian, Ubuntu, Mint, Astra, etc.)
- RPM packages (CentOS, RHEL, Alt, RedOS, etc.)
- tar.bz2 archive for manual installation ("tarball") - for all other Linux systems

Installation from deb (Ubuntu, debian, etc.)

For systems with apt, installation is performed by the following command:

```
sudo apt install -y ./forward-driver_<version>_all.deb ./forward-
utils_<version>_amd64.deb
```

To update, it is enough to install a new version of the package - the old driver will be deleted automatically.

Installation from the tar archive

To install the driver from the archive, unpack it to the root directory, and then add the driver to the dkms:

```
sudo tar -C / -xhvf forward-driver-<version>.tar.bz2
sudo tar -C / -xhvf forward-utils-<version>.tar.bz2
sudo dkms install forward/<version>
```

There is no option to remove the driver in this case, but it can be done manually by removing the driver from dkms and deleting the directories:

```
sudo dkms remove forward/<version>
sudo rm -r /usr/src/forward-<version>
sudo rm -r /lib/firmware/forward
```

Installation checking

After installing the driver in any way, the `/usr/src/forward-<version>` directory with the driver sources should appear in the system. The `modprobe forward` command should also be worked out correctly, and the `dkms status` command should show the presence of the installed drivers in the system. Example of `dkms status` output :

```
forward/<version>, 6.17.9-arch1-1, x86_64: installed
```

Firmware update

Along with the driver, the actual firmware versions for the cards (that are guaranteed to work with installed driver version) are installed in the system (directory `/lib/firmware/forward`). Therefore, it is recommended to check the firmware versions of the boards after driver installation, and possibly update them.

The exception is the FD2110 card - it is reflashed dynamically every time the driver starts, so it does not need a firmware update.

The firmware update is done using the forward-flash utility from the forward-utils package:

```
forward-flash
```

The utility will check the relevance of the firmware and offer to update the firmware, if required. After the update, the utility will prompt you to reboot the board to run the new firmware.

Board reboot mechanism does not work correctly on all systems, especially in the case of virtual machines. If the board does not work or does not work correctly after the update, a cold restart of the computer is required - that is, a complete shutdown and power on (power cycle).

Driver usage guide

After installing and rebooting or running modprobe forward, the device `/dev/forward/<board>-<number>` should appear in the system, for example `/dev/forward/fd722-70802`. This is a service device intended to configure the general parameters of the board, update the firmware, etc. In addition, a directory with the parameters (attributes) of the board should appear in sysfs, some of which can be changed.

sysfs attributres

They are located in the `/sys/class/forward/<board>-<number>/ directory`. From user and other programs point of view, these files are single-line text files, but in fact, they are special pseudo-files that are accessed by calling the service functions inside the driver. With manual configuration, these attributes can be read using cat, and written using echo, for example:

```
cat /sys/class/forward/fd722-70802/temperature
```

will show current board temperature

```
echo ---- > /sys/class/forward/fd722-70802/io_config
```

will turn off all inputs and outputs of the board (more on this below).

Note: Writing attributes to sysfs can only be done from the superuser (root), and there is no good way to give sysfs rights to regular users.

There is some useful sysfs attributes:

- `software_id` *(r)* - logical (serial) number of the boards
- `hardware_id` *(r)* - physical (unchangeable) number of the board
- `temperature` *(r)* - board temperature
- `mode` *(rw)* - the operating mode, is a list of sub-drivers that currently service this board, usually `v4l2,alsa`. The user can change this attribute, for example, to disable ALSA is on the board
- `available_modes` *(r)* - an available set of sub-drivers for the board
- `version` *(r)* - board and firmware version, in the `<board version>r< firmware version>` format
- `io_config` *(rw)* - configurable I/O mode. This attribute contains a number of characters equal to the number of inputs/outputs, each character corresponds to one input/output and can only take the values `I` (input), `O` (output) or `-` (disabled)

IO mode switching

Some (FD788, FD722M2, FD2110) boards have bidirectional interfaces - they can be either inputs or outputs (but not simultaneously). There is an `io_config` attribute for configuring the interface direction in sysfs. It is a string consisting of the characters `I`, `O`, `-`, where each symbol corresponds to one of the board connector. The interfaces are arranged in the order of their physical location on the board: from top to bottom in rackmount case, or from right to left in tower orientation. For example, the default `io_config` line on the FD788 board looks like `IIIIIIII`, i.e. all the interfaces are SDI inputs, and the FD722 has `IIOO`, i.e. there are two inputs first, then two outputs.

Interfaces can have the following states:

- `I` - input
- `O` - output
- `-` - disabled - in this mode, the interface is electrically disabled, the devices do not appear in the

system.

Some boards have fixed inputs and fixed outputs, in this case the interface can only be disabled. Others can be reconfigured:

- FD788 - 8 SDI interfaces, any one can be either input or output. By default all are inputs - `IIIIIIII`
- FD722M2 - 1 fixed input, then 1 configurable interface, then 1 output. Accordingly, `io_config` can be either `IIO` or `IOO`. By default, `IIO` used.
- FD2110 - 2 configurable SDI interfaces, then 16 virtual channels - inputs of the first ethernet, then 16 channels - outputs of the first ethernet, then the same for the second ethernet. In total, the `io_config` string has 66 characters. By default, all interfaces are disabled (`-`) except for the first two SDI, which are configured as inputs (`II`).

Reconfiguration of the interfaces is possible using the echo command. During reconfiguration, all applications using reconfigurated interfaces must be stopped. For example, to set the FD788 board to 4 inputs + 4 outputs, you can run the following command:

```
echo IIII0000 > /sys/class/forward/fd788-80002/io_config
```

You can do the same from your applications by working with attributres same as with files (for example, open(), write() for the C language).

Persist attributes between reboots

The driver system in Linux is stateless - drivers cannot save any state between restarts. This means that after a reboot, all the board settings are reset to their original settings. Settings made in sysfs can be saved as udev rules. These rules allow you to automatically record attributes when a board appears in the system.

For example, to set the FD788 to 4 inputs + 4 outputs at startup, you can create a file `/etc/udev/rules.d/90-fd788.rules` with following content:

```
SUBSYSTEM=="forward", ATTR{software_id}=="80002", ATTR{io_config}:="IIIII000"
```

This rule literally says: "when a device of the forward class appears with the software_id attribute set to 80002, write the value IIIIOOOO to its io_config attribute." Learn more about udev rules

V4L2

The `forward-v4l2` driver provides video access via the standard Video4Linux2 (V4L2) API. This API is supported by many third-party software, for example, OBS Studio, gstreamer, ffmpeg , etc. A `/dev/videoX` device is created for each interface of the board, where X is the video device number in the system. For example, if the computer has one FD788, `/dev/video0` - `/dev/video7` will be created. With two boards, `/dev/video0` - `/dev/video7` for the first board and `/dev/video8` - `/dev/video15` for the second will be created. The numbering of the boards depends on the order of their appearance in the system, which most often corresponds to a certain order of the boards in the PCIe slots, i.e. it does not change when the system is rebooted.

Video setup takes place through standard utilities from the `v4l-utils` package - the `v4l2-ctl` console tool and the `qv4l2` GUI application.

Configuring the video format

In the V4L2 API, the video format ("mode", "mode") is set by two different entities:

- Timing - sets the width, height, and frequency of the signal on the **physical interface**. Can be changed by the user.
- Format - specifies exactly the format of the exchange **between application and driver**. Actual format is negotiated by application and driver.

You can use `v4l2-ctl` to control timings.

Getting a list of available timings:

```
v4l2-ctl -d /dev/videoX --list-dv-timings
```

This command will display a detailed list of timings available for the board, each one is fully described, for example, for 1080i50, the timing will look like this:

```
        Index: 0
        Active width: 1920
        Active height: 1080
        Total width: 2640
        Total height: 1125
        Frame format: interlaced
        Polarities: +vsync +hsync
        Pixelclock: 74250000 Hz (50.00 fields per second)
        Horizontal frontporch: 0
        Horizontal sync: 720
        Horizontal backporch: 0
        Field 1:
        Vertical frontporch: 2
        Vertical sync: 20
        Vertical backporch: 0
        Field 2:
        Vertical frontporch: 3
        Vertical sync: 20
        Vertical backporch: 0
        Standards: SDI
        Flags: CE-video
```

In total, the following timings are available in the boards (with corresponding indices):

1. 1080i50
2. 1080i60
3. 1080i59.94
4. 576i50
5. 480i59.94
6. 720p50
7. 720p60
8. 720p59.94
9. 1080p25
10. 1080p30
11. 1080p29.97
12. 1080p24
13. 1080p23.98
14. 1080p50
15. 1080p60
16. 1080p59.94
17. 720p25
18. 720p30
19. 720p29.97

20. 2160p25
21. 2160p30
22. 2160p29.97
23. 2160p24
24. 2160p23.98
25. 2160p50
26. 2160p60
27. 2160p59.94

Due to the fact that some tamings are not available on some boards or on some interfaces, this list may vary

To set the output timings or force the input timings, use the following command:

```
v4l2-ctl -d /dev/videoX --set-dv-bt-timings index=<index>
```

Where the "index" is the sequence number of the timings in the list obtained using --list-dv-timings, for example 1080p50 usually has the number `13`

If the correct signal is present at the input, you can query its timings:

```
v4l2-ctl -d /dev/video2 --query-dv-timings
```

Or ask the driver to automatically query and set the correct input timings:

```
v4l2-ctl -d /dev/video2 --set-dv-bt-timings query
```

After setting up the timings, the board can work with third-party software.

The timings can also be configured using the `qv4l2` GUI application.

Output mute

By default, the interfaces appear with the outputs electrically disconnected - the signal is transmitted to the output of the boards, but the output itself is turned off and the signal does not appear on the connector. The output turns on automatically at the beginning of image transmission via the V4L2 device, and the device turns off again at the end of the application. But you can manually turn on/off the output using `v4l2-ctl`

```
v4l2-ctl -d /dev/videoX -c mute_output=0
```

It will turn on the output, and the board will transmit a black image until the any application starts transmission.

After the first installation of mute_output, the automatic on/off will stop working.

Genlock setup

In boards with SDI outputs, it is possible to manually adjust the frequency and synchronize the outputs. All boards have one common clock generator, from which all outputs operate. This generator can be operated either from its internal source (master) or synchronized to an analog video signal on a separate analog input. Synchronization to external analog signal is called an analog genlock, blackburst sync or tri-level sync. Also, in the master mode, it is possible to manually adjust the frequency of the generator within ± 500ppm (± 0.05%) - this allows synchronization to external sources, such as NTP.

In addition to the analog genlock common to the entire board, each output can be independently synchronized to any input. This is called a digital genlock, in this mode, the output starts operating at the frequency of the

input SDI, and, accordingly, asynchronously to the rest of the outputs. This mode allows you to transfer video from input to output without dropping/inserting frames (without time-based correction).

The genlock is configured via the controls of the output v4l2 device using the `v4l2-ctl` utility or the V4L2 API

```
v4l2-ctl -d /dev/videoX -c genlock_source=1
```

Will set the synchronization source for the output (or the entire board). Possible options:

1. Master
2. Analog genlock input
3. First SDI-input
4. Second SDI-input
5. ...

After setting genlock source, you can view the current status of the genlock:

```
v4l2-ctl -d /dev/videoX -C genlock_state
```

Possible states:

1. Master - exit in master mode, genlock is not enabled
2. No input signal - there is no input signal for the genlock, or it is in the wrong format (the frame rate does not match the output)
3. Locking - the output adjusts its frequency to the source
4. Locked - the output is adjusted to the source and synchronized
5. Holdover - the output was once synchronous, but then the input signal disappeared, the output operates at the last locked frequency

In the genlock mode, each output can be phase shifted relative to the synchronization source within ±frame/2 period with an accuracy of 6.734ns (1pixel, 1/148500000s). This is also done through controls:

```
v4l2-ctl -d /dev/videoX -C genlock_offset_6_734ns=1000000
```

The command above will shift the image by 6.734ms into the future, i.e. the output frame will be 6.734ms late relative to the input.

Other V4L2 settings

The full list of controls available to the board can be viewed with the following command:

```
v4l2-ctl -d /dev/videoX -l
```

Get the control value:

```
v4l2-ctl -d /dev/videoX -C enable_vanc
```

Set the control value:

```
v4l2-ctl -d /dev/videoX -c enable_vanc=1
```

Also, the `qv4l2` application allows you to do all this from the graphical interface.

Some of the controls are service or intended for use from applications, but there are useful user controls:

- `analog_input_mode` - is the mode of operation of the analog input (genlock input), 0 - blackburst/tri-level/VITC receiving, 1 - PPS pulses receiving, 2 - LTC receiving
- `bypass_disable_forced` - for the FD722BP card, forcibly turn off the hardware bypass
- `enable_vanc` - turn on the reception of VBI data, after that the `/dev/vbiX` devices begin to work correctly
- `frequency_adjust_ppt` - change the carrier frequency for video transmission
- `genlock_source` - is the synchronization source for the output. Maybe `Master (0)`, `Analog (1)` or specific `SDI (N + 1)`
- `genlock_state` - the current synchronization status. There can be `Master (0)`, `No input (1)`, `Locking (2)`, `Locked (3)`, `Holdover (4)`
- `genlock_offset_6_734ns` - output offset relative to the sync source, in 3G-SDI pixels (1/148500000 s).
- `cloned_output` - enable signal cloning. The even (2, 4, 6, 8) outputs of the FD722, FD788, and FD922 boards can enter cloning mode when a they are copy of the signals of outputs 1, 3, 5, and 7.
- `mute_output` - turn it off electrically ("turn it off") exit. At the same time, the V4L2 device itself and the board continue to work, but electrical idle (Hi-Z) remains on the interface.

For more information about V4L2 controls, see the SDK section.

ALSA

The `forward-alsa` driver provides access to audio through the standard ALSA API. This API is supported by many third-party software, for example, pulseaudio, OBS Studio, gstreamer, ffmpeg, etc. An ALSA device `hw:Y,X` is created for each interface of the board, where Y is the serial number of the card (card in ALSA) in the system, X is the output number (device in ALSA). Unlike V4L2, ALSA devices always have a fixed X number, depending only on the physical location on the board. For example, if there is one FD788 in the computer and there are no other ALSA cards, `hw:0,0` - `hw:0,7` will be created. With two boards, `hw:0,0 - hw:0,7` for the first board and `hw:1,0 - hw:1,7` for the second will be created. At the same time, the numbering of the boards depends on the order in which they appear in the system, which most often corresponds to a certain order of the boards in the slots, i.e. it does not change when the system is rebooted.

Audio setup is not necessary, audio testing can be done through standard utilities from the `alsa-utils` package.

To get a list of input devices, you can use the `arecord` utility:

```
arecord -l
```

Typical ouput:

```
card 0: Intel [HDA Intel], device 0: Generic Analog [Generic Analog]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: F80002 [FD788 80002], device 0: IN [IN 1]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: F80002 [FD788 80002], device 1: IN [IN 2]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: F80002 [FD788 80002], device 2: IN [IN 3]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: F80002 [FD788 80002], device 3: IN [IN 4]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: F80002 [FD788 80002], device 4: IN [IN 5]
  Subdevices: 1/1
```

```
  Subdevice #0: subdevice #0
card 1: F80002 [FD788 80002], device 5: IN [IN 6]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: F80002 [FD788 80002], device 6: IN [IN 7]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
card 1: F80002 [FD788 80002], device 7: IN [IN 8]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

Here `hw:0,0` is the built-in system sound card, `hw:1,0` - `hw:1,7` is the FD788 card with the number 80002.

To get a list of output devices, you can use the `aplay` utility:

```
aplay -l
```

Pulseaudio/pipewire

Since ALSA devices are the standard way to work with sound in Linux, standard system components begin to consider FD boards as ordinary sound cards and try to use them as ordinary speakers for audio output and microphones for capture. Along with the drivers, a set of alsa profiles (/usr/share/alsa-card-profile) is installed that prevent this behavior, however, sometimes (especially if the card is the only sound device) pulseaudio still grabs the device and prevents the rest of the software from working. In this case, you need to find the sound section in the system settings and set the "Disabled" profile for the FD card.

gstreamer usage examples

gstreamer is a multimedia framework that works in terms of pipelines (pipelines, graphs), allowing for real-time video processing. Working with the forward driver is possible through the standard components `v4l2src`, `v4l2sink`, `alsasrc`, `alsasink`.

The only limitation is that the alsa driver uses a v4l2 device as synchronization, and gstreamer makes them reference clocks by default when using alsasink/alsasrc. In some graphs, this may leads to deadlock, so in this case, the provide-clock=false parameter must be set for the alsa components.

Capture

Video and audio capture and display/system speakers:

```
v4l2-ctl -d /dev/videoX --set-dv-bt-timings query
gst-launch-1.0 v4l2src device=/dev/videoX ! queue ! autovideosink alsasrc
device=hw:Y,X ! queue ! autoaudiosink
```

Video and audio capture and encoding to a file:

```
v4l2-ctl -d /dev/videoX --set-dv-bt-timings query
gst-launch-1.0 -e v4l2src device=/dev/videoX ! queue ! videoconvert ! x264enc
! queue ! mux. \
               alsasrc device=hw:Y,X ! queue ! audioconvert ! opusenc ! queue
! mux. \
               mp4mux name=mux ! filesink location=/tmp/capture.mp4
```

Playback

Test video output:

```
gst-launch-1.0 videotestsrc ! "video/x-raw,interlace-mode=interleaved" !
queue ! v4l2sink device=/dev/videoX audiotestsrc ! queue ! alsasink
device=hw:Y,X
```

"video/x-raw,interlace-mode=interleaved" is needed here only for interlaced formats. For some reason, videotestsrc cannot detect interlacing automatically.

Video pass-through processing

For example, take the video from the first input of the FD722, overlay the text with the stream time on it and send it to the first output. At the same time, turn on the genlock output to the first input:

```
v4l2-ctl -d /dev/video2 -c genlock_source=2
gst-launch-1.0 v4l2src device=/dev/video0 ! queue ! timeoverlay ! queue !
v4l2sink device=/dev/video2 \
               alsasrc provide-clock=false device=hw:0,0 ! queue ! alsasink
provide-clock=false device=hw:0,2
```

At the first output of the FD722, a signal will appear from the input with the time overlay.

SDK

The forward driver implements the standard V4L2 and ALSA API with some additional functionality. Therefore, the SDK consists of examples of using these APIs. The V4L2 API documentation is available at kernel.org Communication with the ALSA subsystem takes place through the libasound library, the documentation is available on alsa-project.org Everything necessary for non-standard functionality is defined in the header file `/usr/src/forward-<version>/include/forward-v4l2-ioctl.h`

List of examples

An archive with examples of using the V4L2 API with the forward driver is supplied with the driver. The examples are written in C++.

- `common` - is a small library with common functions, including a C++ wrapper for a V4L2 device
- `enumerate` - is an example of using libudev for forward device search and mapping v4l2 devices with ALSA
- `io-switch` - an example of switching the direction of outputs (writing to sysfs) from the code
- `timings` - an example of working with timings
- `genlock` - example of enabling genlock
- `capture-simple` - example of video capture
- `playback-simple` - example of video playback
- `capture-vbi-audio` - example of video, vbi and audio capture
- `playback-vbi-audio` - example of video, vbi and audio playback
- `capture-complex` - example of video, vbi and audio capture via a single v4l2 interface
- `playback-complex` - an example of video, vbi and audio playback via a single v4l2 interface
- `capture-fd2110` - an example of video capture over a network for an FD2110 board
- `playback-fd2110` - example of video playback over the network for the FD2110
- `capture-raw` - example of capturing a "raw" SDI stream to a file
- `playback-raw` - an example of playing a "raw" SDI stream from a file
- `capture-preview` - an example of capturing video and displaying it in a Qt window
- `playback-watchdog-bypass` - example of video playback using a board FD722BP and watchdog timer
- `passthrough-simple` - example of video passing from input to output with processing
- `av-delay-generator` - test signal generator to check the sync between video and audio
- `av-delay-viewer` - viewing the test signal to check the sync between video and audio
- `hw-timestamps` - is an example of getting timestamps based on the clock of the board
- `analog-pps` - is an example of receiving timestamps based on the board clock from the PPS signal at the analog input
- `analog-tc` - example of receiving a timecode from the analog input VITC or LTC
- `capture-asi-raw` - example of capturing a "raw" ASI stream, decoding and writing to a file
- `playback-asi-raw` - example of playing a "raw" ASI stream with encoding from a file
- `passthrough-asi` - example of capturing a "raw" ASI stream, decoding and transmitting to the output

How V4L2 works

Communication with the driver takes place through the device files `/dev/videoX` via the `ioctl()` system call. The operation code (`VIDIOC_*`) and a pointer to the structure through which information is exchanged are passed to the `ioctl()` call. Through `ioctl()` , the timing and format are configured, controls changed, buffers allocated, etc. The `forward` driver since version `1.7.0` uses the multiplanar Video4Linux2 API.

Video input and output happens in the following order:

1. Opening the device using `open()`
2. Requesting and setting the input/output timing using ioctl `VIDIOC_ENUM_DV_TIMINGS`, `VIDIOC_QUERY_DV_TIMINGS`, `VIDIOC_S_DV_TIMINGS`
3. Request, setting the format for communicating with the driver - `VIDIOC_ENUM_FMT`, `VIDIOC_G_FMT`, `VIDIOC_S_FMT`
4. Buffer allocation using `VIDIOC_REQBUFS`, `VIDIOC_QUERYBUF`
5. Pre-filling the buffer queue (pre-roll) - `VIDIOC_QBUF`
6. Enabling playback - `VIDIOC_STREAMON`
7. The cycle of receiving buffers from the queue, processing, and transferring them back to the queue - `VIDIOC_DQBUF`, `VIDIOC_QBUF`
8. Stopping playback - `VIDIOC_STREAMOFF`
9. Buffer freeing - `VIDIOC_REQBUFS`

Timings and formats

In V4L2, there are two entities that determine the video parameters:

- *Timing* - determines the parameters of the signal at the input / output of the board - the size, frequency, and size of the blanking intervals.
- *Format* - defines the parameters of the video in the buffers - size, pixel format, stride, etc.

The timing is described by the `v4l2_bt_timings` structure, which specifies the geometric dimensions, the visible part, the size of the sync regions, the carrier frequency, etc. To get a list of all possible timings, you need to list them using ioctl `VIDIOC_ENUM_DV_TIMINGS`. The timings index is passed to this ioctl in turn, starting from 0, until the ioctl returns the -EINVAL code. For input, it is also possible to request the current time at the entrance using `VIDIOC_QUERY_DV_TIMINGS`.

Both of these ioctls return a timings structure, which can then be passed to `VIDIOC_S_DV_TIMINGS` for setting the signal parameters. Calling `VIDIOC_G_DV_TIMINGS` gets the current timings on the device (the one that was set using `VIDIOC_S_DV_TIMINGS`, not the one that is actually on the input). The input device does not update the timings when the input signal changes, this must be done manually. After streaming start (`VIDIOC_STREAMON`), the timing cannot be changed.

The format is described by the `v4l2_pix_format_mplane` structure. It describes the size of the video, the pixel format, the size of the plains, the method of transmitting fields, etc. The pixel format is a 4-byte constant, better known as FourCC, which is also a unique format identifier. In the case of interlaced timings, format also allows you to choose how to store fields in the buffer:

- `V4L2_FIELD_INTERLACED` - both fields are stored in the buffer interlaced, the first line it is the first line of the first field
- `V4L2_FIELD_ALTERNATE` - only one field is stored in the buffer, the frame consists of two buffers

In the current implementation, V4L2_FIELD_INTERLACED requires additional data copying, while V4L2_FIELD_ALTERNATE is zero-copy

Calling `VIDIOC_ENUM_FMT` works on the same principle as `VIDIOC_ENUM_DV_TIMINGS` and returns a list of FourCC formats available for a given timings. To set the format you must first get it (`VIDIOC_G_FMT`), then change the fields in the structure (usually, only FourCC), then set it (`VIDIOC_S_FMT`). The API is designed in such a way that `VIDIOC_S_FMT` is considered a recommendation to the driver, it has the right to adjust the format and return the modified format to the application.

The `forward` driver implements both standard formats (YUYV, V210, etc.) and has a set of specialized formats (Raw, ASI, Complex) for advanced functions.

Buffers and queues

The video itself is transmitted in terms of queues and buffers. A buffer is an entity containing a single frame (or

field), allocated in the kernel via the `ioctl()` call and mapped (`mmap()`) into the application's address space. The buffer can have one or more planes, each plane is a continuous (virtually) area in memory, of a specific (by format) size. In simple cases, there is only one plane and it contains an image in a certain format, for example, in the YUYV 8-bit format (V4L2_PIX_FMT_YUYV) a single plain will contain YUV pixels in the 4:2:2 format, 16 bits per pixel.

In addition to the image itself, buffers store some meta-information:

- Sequence number - a frame counter for tracking losses
- Timestamp - the system arrival time/sending a frame
- Timecode (VITC/LTC)
- In some formats, each buffer also indicates how much data is actually used in each play (bytesused)

Buffers are allocated via ioctl `VIDIOC_REQBUFS` , and the desired number of buffers (frames) is passed in the parameters. Each buffer has its own unique identifier, a sequence number, and its index passed as a buffer parameter in all calls. After allocating the buffers, you need to map buffers onto the application's address space and get pointers to the data in the plains. To do this:

- call `VIDIOC_QUERYBUF` for each buffer to get detailed information about the number of planes, their size, and offset for mmap.
- Make `mmap()` call for each plain of each buffer, similar to shared memory, to access data

After that, the buffers are ready for use in the driver, while the application also has access to the video data. To release the buffers, you need to execute `VIDIOC_REQBUFS` by passing `0` as the number of buffers.

Data exchange with the driver and the board takes place in terms of queues. Each device has a queue of buffers, the application adds a buffer to the queue using ioctl `VIDIOC_QBUF` . After that, the buffer comes under the control of the driver and, as the buffers processed, they are transferred to the output (input) of the board. After the board has finished working with the buffer, it can be returned by the application. For this, the ioctl `VIDIOC_DQBUF` is used - it is a blocking call waiting for the first processed buffer from the driver. When opening the device in non-blocking mode (parameter `O_NONBLOCK` for open()) `VIDIOC_DQBUF` is not blocked, but returns an error if the buffer is not ready. In this case, the `poll()` system call have to be used. After `VIDIOC_DQBUF` , the data is ready for processing by the application.

In total, data input/output using queues looks like this:

1. Buffer allocation (`VIDIOC_REQBUFS`)
2. Buffer preparation (`VIDIOC_QUERYBUF`)
3. Pre-filling the queue (preroll) (`VIDIOC_QBUF`)
4. Stream start (`VIDIOC_STREAMON`)
5. In cycle:
    1. Waiting for the buffer to be ready (`poll()`)
    2. Receiving the buffer from the driver (`VIDIOC_DQBUF`)
    3. Data processing
    4. Returning the buffer to the queue (`VIDIOC_QBUF`)
6. Stream sop (`VIDIOC_STREAMOFF`)
7. Buffer freeing (`VIDIOC_REQBUFS`)

For the correct operation of the boards, it is required that there are always at least 2 buffers in the queue - the board works with one, the second is prepared in advance. This means that it takes at least 3 buffers to work

If there is no buffer in the queue to transfer to the board at the end of the frame, a frame loss event occurs. The frame is skipped for input, the last buffer is taken for output (if available) and repeated. You can determine frame loss using the `sequence` field of the buffers - this is the frame counter on the board. If the sequence has changed by more than 1 after the last `VIDIOC_DQBUF` , this means that a certain number of frames have been lost.

VBI

Together with the `/dev/videoX` video devices, the driver creates standard VBI `/dev/vbiX` devices. The VBI (Vertical Blanking Interval) area is transmitted through these devices. Working with them is similar to working with `/dev/videoX` - setting the format, allocating buffers, queues. The format is the `v4l2_vbi_format` structure, which transmits the number of lines and the pixel format for VBI. In current boards, the pixel format for VBI is `V4L2_PIX_FMT_Y10` - all 10-bit pixels are packed into 16-bits, with the highest part zeroed. Explicit control activation is required for VBI to work - use `V4L2_CID_FORWARD_ENABLE_VANC` control at the `/dev/videoX` device.

Raw mode

In boards with SDI interfaces, it is possible to receive and output the entire 10-bit SDI stream, with all synchronization areas and packets. This can be used to analyze the SDI stream, record it, and then play it back, as well as for advanced video processing with minimal content modification. The raw mode is set by setting the pixelformat format = `V4L2_PIX_FMT_SL_RAW` on the V4L2 device.

After setting the format, the raw stream parameters - full width, height and size in bytes will be described in the format structure. For interlaced formats, the raw mode can only work in `V4L2_FIELD_ALTERNATE` , i.e. each field will be in a separate buffer. The data itself in the buffers is in the form of 10-bit words, tightly packed in 4 words in 5 bytes (40 bits). The order of the SDI streams is first CbCr, then Y. The word order is little-endian, i.e. the first byte contains the least significant 8 bits from the first received a 10-bit word:

Byte\Bit	7							0
0	Cb0[7:0]							
1	Y0[5:0]						Cb0[9:8]	
2	Cr0[3:0]				Y0[9:6]			
3	Y1[1:0]		Cr0[9:4]					
4	Y1[9:2]							

Since the stream is transmitted to the output "as is", the task of the software is to correctly generate all the required markers of the SDI stream - SAV, EAV, and so on. Any disturbance in the stream (for example, the luminance/color value outside the range 0x004 - 0x3FB) will result in loss of signal at the receiver. The `capture-raw` and `playback-raw` examples show an example of capturing a stream to a file and playing it from a file, respectively.

ASI

Cards with SDI interfaces can receive and output DVB ASI stream. The stream can be received in two formats at the input:

- `V4L2_PIX_FMT_MPEG` - each buffer will contain decoded MPEG-TS packets. Field `bytesused` will indicate the number of bytes received.
- `V4L2_PIX_FMT_SL_RAW_ASI` - each buffer will contain a full undecoded 8b10b stream with all K-characters. The packing is the same as in raw mode - 4 words in 5 bytes. Field `bytesused` will indicate the number of bytes.

Only the `V4L2_PIX_FMT_SL_RAW_ASI` format is possible at the output, while 10-bit words are not 8b10b encoding, but an internal format in which bits [7:0] are data, and bit [8] indicates whether the word is a K-character (`1`) or data (`0`).

When an ASI signal is detected at the input, the driver automatically sets the format `V4L2_PIX_FMT_MPEG` - this way it is possible to detect the presence of an ASI signal via `VIDIOC_G_FMT` . If ASI has switched to SDI at the input, the driver automatically returns the format `V4L2_PIX_FMT_YUYV` .

When receiving/transmitting ASI, the boards operate in 20ms intervals (bursts), that is, each buffer corresponds to a stream time of 20ms or 540,000 characters. To reduce delays, it is possible to reduce the interval up to 2.5ms through the `V4L2_CID_FORWARD_ASI_PERIOD` control.

Complex mode

Due to the fact that the ALSA and V4L2 devices are virtually independent of each other, the exact Audio synchronization with sound is not a trivial task, especially in the case of problems with the input signal and sound loss. To simplify synchronization, a non-standard mode was implemented for V4L2 - in the form of `V4L2_PIX_FMT_SL_COMPLEX_8BIT` and `V4L2_PIX_FMT_SL_COMPLEX_10BIT` formats. The frame (field) in this format consists of three planes:

1. The video itself, 8 bits UYVY (`V4L2_PIX_FMT_SL_COMPLEX_8BIT`) or dense packed 10 bits UYVY (`V4L2_PIX_FMT_SL_COMPLEX_10BIT`).
2. Input - all received SDI/HDMI/RTP packets per frame, including audio. The output is the audio corresponding to the frame, in the stream format.
3. The `forward_v4l2_complex_info` structure, which describes the current frame and statistics, is filled in by the driver.

In this mode, the driver does not process the data - it processed to the application/board in the format that the board uses. For interlaced formats, the complex mode can only work in `V4L2_FIELD_ALTERNATE` , i.e. each field will be in a separate buffer. For example, if VBI is enabled (via the `V4L2_CID_FORWARD_ENABLE_VANC` control), then the VBI area in 10-bit format will always be at the beginning of the data plane. The size of this area can be calculated using timing (`.bt.vsync` + `.bt.vbackporch`), and on the input device - in the `forward_v4l2_complex_info` structure.

To understand what format to expect in the second play, you need to use the control `V4L2_CID_FORWARD_COMPLEX_AUDIO_FORMAT` :

- `V4L_FORWARD_COMPLEX_AUDIO_RAW_32` (0) - 24 bits of audio in 4 bytes, bits in the lower part (little-endian). It is used for the output of SDI cards.
- `V4L_FORWARD_COMPLEX_AUDIO_RAW_24BE` (1) - 24 bits in 3 bytes, Big-endian, used in the output of FD2110
- `V4L_FORWARD_COMPLEX_AUDIO_ST272` (2) - Audio in packets according to the ST-272 standard (SD-SDI audio). Each word in the buffer is 16 bits, containing 10-bit packet data according to the standard SMPTE ST-291.
- `V4L_FORWARD_COMPLEX_AUDIO_ST299` (3) - Audio in packages according to the ST-299 standard (HD-SDI audio). Each word in the buffer is 16 bits, containing 10-bit packet data according to the standard SMPTE ST-291.
- `V4L_FORWARD_COMPLEX_AUDIO_HDMI` (4) - Audio in packets according to the HDMI standard
- `V4L_FORWARD_COMPLEX_AUDIO_ST2110_30` (5) - Audio in packages according to the ST2110-30 standard
- `V4L_FORWARD_COMPLEX_AUDIO_AES3` (6) - 4-byte aes frames, used in FD788 when receiving AES3id.
- `V4L_FORWARD_COMPLEX_AUDIO_ST2022_6` (7) - Audio in packages according to the ST2022-6 standard

For packet-based formats, the common library includes appropriate parsers that can extract raw pcm audio samples. It is worth noting that the formats `V4L_FORWARD_COMPLEX_AUDIO_ST272` and `V4L_FORWARD_COMPLEX_AUDIO_ST299` , in addition to audio, allows you to receive any SMPTE ST-291 packets from the SDI stream.

Events

The driver sends events via the standard V4L2 interface (API). The driver sends the following events:

- `V4L2_EVENT_SOURCE_CHANGE` - is a standard format change event
- `V4L2_EVENT_VSYNC` - is a standard synchronization event that is sent once per field, regardless of whether playback is enabled or not.
- `V4L2_EVENT_FRAME_SYNC` - is a standard synchronization event that is sent once per frame

before transferring the buffer to the application when playback is enabled. Contains the current frame number.
- `V4L2_EVENT_FORWARD_TIMESTAMP` - is a driver-specific event that contains the current frame timestamp and is sent once per field, regardless of whether playback is enabled or not. The event passes the struct `v4l2_event_forward_timestamp` structure with the following fields:
  - `buffer_sequence` - the current frame number, matches the buffer and `V4L2_EVENT_FRAME_SYNC`
  - `field` - the current field, `V4L2_FIELD_*`
  - `eof_timestamp` - hardware timestamp of the field/frame, the format is the same as in `V4L2_CID_FORWARD_HW_TIMESTAMP`
  - `irq_timestamp` - hardware interrupt timestamp - the time when the information reached the driver, the format is the same as in `V4L2_CID_FORWARD_HW_TIMESTAMP`

V4L2 controls list

These controls are available via the V4L2 Standard interface (API) and can be changed using standard v4l2-ctl and qv4l2. The header file `/usr/src/forward-<version>/include/forward-v4l2-ioctl.h` contains all the necessary declarations for using these controls in your code. Most controls are available for both inputs and outputs. Also, most of them can be changed while receiving/transmitting the stream, without stopping. Some controls are common to the entire board - when you change them in one device, they change for all devices.

- `V4L2_CID_FORWARD_ENABLE_VANC` (`bool`) - enable capture/playback of the VANC area **only when streaming is stopped**
- `V4L2_CID_FORWARD_WIDESCREEN` (`bool`) - the input signal has the widescreen flag in the VPID / set the widescreen flag on the output
- `V4L2_CID_FORWARD_HW_TIMESTAMP` (`u32`) - get the hardware time. The boards have a counter that operates at a frequency of 148.5MHz, synchronously with the outputs - an hardware clock. **read-only**
- `V4L2_CID_FORWARD_FREQUENCY_ADJUST` (`u64`) - adjusting the output frequency in ppb (point-per-trillion, 10^-12). By writing values to this control, you can smoothly (without losing the output signal) change the carrier frequency at the output - `Fout = V4L2_CID_FORWARD_FREQUENCY_ADJUST/1e-12 * Fbase` . It is not recommended to change the value by more than 10ppm (1e-6) at a time, as this may lead to signal loss on inputs and outputs for a few seconds. The control limit is ±500ppm **common for the entire board**
- `V4L2_CID_FORWARD_ASI_PERIOD` (`enum`) - switch the period of data (buffers) from the driver for ASI **only when streaming is stopped**
  - `20ms (0)` - *default*
  - `10ms (1)`
  - `5ms (2)`
  - `2.5ms (3)`
- `V4L2_CID_FORWARD_TIMECODE` (`enum`) - type of timecode to receive at the input/type of timecode at output
  - `None (0)` - no timecode - *default*
  - `LTC (1)`
  - `VITC (2)`
  - `Any (3)`
- `V4L2_CID_FORWARD_GENLOCK_SOURCE` (`enum`) - genlock source, carrier frequency locking.
  - `V4L_FORWARD_GENLOCK_SRC_MASTER` (0) - is the master, the carrier frequency does not depend on the inputs, it can be changed using `V4L2_CID_FORWARD_FREQUENCY_ADJUST` , *default*
  - `V4L_FORWARD_GENLOCK_SRC_ANALOG` (1) - lock the carrier frequency to the analog input (black-burst/tri-level-sync) **common for the entire board**
  - `V4L_FORWARD_GENLOCK_SRC_IN0` (2) - lock the carrier frequency to the first SDI input

    - `V4L_FORWARD_GENLOCK_SRC_IN1` (3) - lock the carrier frequency to the second SDI input
    - `V4L_FORWARD_GENLOCK_SRC_INX` (1 + X) - lock the carrier frequency to the Xth SDI input
- `V4L2_CID_FORWARD_GENLOCK_STATE` (`enum`) - the current state of the genlock **read-only**
    - `V4L_FORWARD_GENLOCK_MASTER` (0) - in master mode
    - `V4L_FORWARD_GENLOCK_NO_INPUT_SIGNAL` (1) - input signal not detected
    - `V4L_FORWARD_GENLOCK_LOCKING` (2) - frequency and phase adjustment is underway
    - `V4L_FORWARD_GENLOCK_LOCKED` (3) - output in genlock, everything is fine
    - `V4L_FORWARD_GENLOCK_HOLDOVER` (3) - the board was in the genlock, and then the signal disappeared, the board is trying to hold the frequency
- `V4L2_CID_FORWARD_GENLOCK_OFFSET` (`u32`) - offset of the output frame relative to the genlock source. Unit - 6.734ns = 1/148.5MHz = 1 pixel of 1080p50/p60 format **for output only**
- `V4L2_CID_FORWARD_ANALOG_RX_MODE` (`enum`) - the operation mode of the analog input of the board **common for the entire board**
    - `Genlock (0)` - analog video synchronization - *default*
    - `PPS (1)` - reception of the PPS (1Hz)
    - `LTC (1)` - reception of the LTC timecode
- `V4L2_CID_FORWARD_ANALOG_RX_TIMESTAMP` (`struct`) - returns the current state analog input. The `v4l2_forward_analog_rx_timestamp` structure is returned, which contains an indication of the presence of a signal at the input, the timestamp of the last event, the current time of the board and the system. **common for the entire board read-only**
- `V4L2_CID_FORWARD_ANALOG_RX_TIMECODE` (`struct`) - returns the current timecode on the analog input, returns the `struct v4l2_timecode` . **common for the entire board read-only**
- `V4L2_CID_FORWARD_BYPASS_DISABLE` (`bool`) - disable hardware signal bypass for the FD722BP board. This control must be written every `V4L2_FORWARD_BYPASS_WATCHDOG_PERIOD_MS` milliseconds at least.
- `V4L2_CID_FORWARD_BYPASS_DISABLE_FORCE` (`bool`) - disable hardware signal bypass for the FD722BP board forever.
- `V4L2_CID_FORWARD_WATCHDOG_ENABLE` (`bool`) - enable hardware shutdown of output when there is no writes to `V4L2_CID_FORWARD_WATCHDOG_KEEP_ALIVE` .
- `V4L2_CID_FORWARD_WATCHDOG_KEEP_ALIVE` (`u32`) - delay output shutdown by X milliseconds.
- `V4L2_CID_FORWARD_CLONED_OUTPUT` (`bool`) - clone output, works only on even outputs for SDI cards.
- `V4L2_CID_FORWARD_MUTE_OUTPUT` (`bool`) - turn off the output electrically, the output itself will continue work, but there will be no signal on the connector.
- `V4L2_CID_FORWARD_COMPLEX_AUDIO_GROUPS` (`u32`) - the number of audio groups (audio channels * 4) in complex mode.
- `V4L2_CID_FORWARD_COMPLEX_AUDIO_FORMAT` (`enum`) - audio type for complex mode **read-only**
    - `V4L_FORWARD_COMPLEX_AUDIO_RAW_32` (0)
    - `V4L_FORWARD_COMPLEX_AUDIO_RAW_24BE` (1)
    - `V4L_FORWARD_COMPLEX_AUDIO_ST272` (2)
    - `V4L_FORWARD_COMPLEX_AUDIO_ST299` (3)
    - `V4L_FORWARD_COMPLEX_AUDIO_HDMI` (4)
    - `V4L_FORWARD_COMPLEX_AUDIO_ST2110_30` (5)
    - `V4L_FORWARD_COMPLEX_AUDIO_AES3` (6)
    - `V4L_FORWARD_COMPLEX_AUDIO_ST2022_6` (7)

Additional controls are provided for the FD2110 board:

- `V4L2_CID_FORWARD_FD2110_STREAM_TYPE` (`enum`) - video stream type, either

`V4L_FORWARD_FD2110_STREAM_ST2110_20` or `V4L_FORWARD_FD2110_STREAM_ST2022_6`
- `V4L2_CID_FORWARD_FD2110_STREAM_BPC` (`enum`) - the number of bits per pixel in the ST-2110 stream (not in v4l2 format!), may be `V4L_FORWARD_FD2110_STREAM_BPC_8BIT` or `V4L_FORWARD_FD2110_STREAM_BPC_10BIT`
- `V4L2_CID_FORWARD_FD2110_STREAM_COLOR` (`enum`) - pixel format in the ST-2110 stream (not in v4l2 format!), so far there can only be `V4L_FORWARD_FD2110_STREAM_COLOR_YCBCR422`
- `V4L2_CID_FORWARD_FD2110_STREAM_ADDRESS_STRING` (`string`) - is the address of the stream in the `ipv4:port` or `[ipv6]:port` format, it can be a multicast or unicast address.
- `V4L2_CID_FORWARD_FD2110_STREAM_ADDRESS` (`struct`) - same as `V4L2_CID_FORWARD_FD2110_STREAM_ADDRESS_STRING` but as binary struct `v4l2_forward_stream_address`
- `V4L2_CID_FORWARD_FD2110_STREAM_SOURCE_PORT` (`u16`) - for output, outgoing port
- `V4L2_CID_FORWARD_FD2110_STREAM_PT` (`u8`) - for output, Payload Type field for RTP packets
- `V4L2_CID_FORWARD_FD2110_STREAM_SSRC` (`u32`) - for output, SSRC field for RTP packets
- `V4L2_CID_FORWARD_FD2110_ST2022_7_TX_ENABLE` (`bool`) - for output, enable stream duplication according to the ST2022-7 standard.
- `V4L2_CID_FORWARD_FD2110_STREAM_AUDIO_ADDRESS_STRING` (`string`) - same as `V4L2_CID_FORWARD_FD2110_STREAM_ADDRESS_STRING` but for audio in complex mode
- `V4L2_CID_FORWARD_FD2110_STREAM_AUDIO_ADDRESS` (`struct`) - same as `V4L2_CID_FORWARD_FD2110_STREAM_ADDRESS` but for audio in complex mode
- `V4L2_CID_FORWARD_FD2110_STREAM_AUDIO_SOURCE_PORT` (`u16`) - same as `V4L2_CID_FORWARD_FD2110_STREAM_SOURCE_PORT` but for audio in complex mode
- `V4L2_CID_FORWARD_FD2110_STREAM_AUDIO_PT` (`u8`) - same as `V4L2_CID_FORWARD_FD2110_STREAM_PT` but for audio in complex mode
- `V4L2_CID_FORWARD_FD2110_STREAM_AUDIO_CHANNELS` (`u8`) - number of audio channels in complex mode

AES3id receiving

The FD788 board allows you to receive an AES3 stream via a coaxial cable (AES3id). No configuration is required for this - the board automatically detects the AES3id stream. In this case, the V4L2 device stops working, and the board can be used as a regular sound card with 8 inputs.

Working with FD2110

The FD2110 board is significantly different from all the others - it has a network interface in addition to video interfaces. The video interfaces themselves require additional configuration to work over the network. In the system, the FD2110 board appears as:

- Two standard network interfaces that can be used as regular network cards
- Two V4L2 devices and two ALSA devices representing an SDI input/output (as in FD788)
- Up to 64 V4L2 devices and 64 ALSA devices representing streams ST2110-20 and ST2110-30 or ST2022-6

Network adapter

The board provides two independent ethernet interfaces (usually enpsd1 and enpsd2). They work as full-fledged network interfaces - they set network settings, speeds, SFP settings, etc., you can transfer data. The only limitation is that each interface has only one packet queue, so performance is limited to a few gigabits (up to 5). **The 2110 video streams normally do not reach these interfaces.**

V4L2 and ALSA

The board provides up to 66 V4L2 and ALSA devices. As with other boards, enabling streams and changing the SDI direction is configured via an attribute in sysfs.

```
/sys/class/forward/fd2110-<number>/io_config
```

For example, by default, `cat /sys/class/forward/fd2110-*/io_config` will return string with 66 characters:

```
II----------------------------------------------------------------
```

Here, each character is one of the inputs/outputs or streams, `-` - the stream is off, `I` is the input, `O` is the output. The streams have the following order:

1. 2 SDI streams, can be `I`, `O`, `-`
2. 16 input streams of the first (furthest from PCIe slot) network interface, can be either `I`, or `-`
3. 16 output streams of the first network interface, can be either `O`, or `-`
4. 16 input streams of the second network interface, can be either `I`, or `-`
5. 16 output streams of the second network interface, can be either `O`, or `-`

When writing a line to `io_config` the corresponding V4L2 `/dev/videoX` and ALSA `hw:X,Y` devices will appear. For example

```
echo III---------------O---------------------------------------------- >
/sys/class/forward/fd2110-*/io_config
```

will turn on all SDI inputs, one input and one output stream on the first network interface, respectively, devices will appear in the system:

- /dev/video0, /dev/video1 - SDI video inputs.
- /dev/video2 - ST2110-20/ST2022-6 video input on the first network interface
- /dev/video3 - ST2110-20/ST2022-6 video output on the first network interface
- "hw:0,0", "hw:0,1" - SDI audio input
- "hw:0,2" - ST2110-30/ST2022-6 audio input on the first network interface
- "hw:0.18" - ST2110-30/ST2022-6 audio output on the first network interface,

Further operation proceeds as with conventional V4L2 and ALSA devices.

Board setup

Network settings (speed, IP, domain, DNS, etc.) are configured by standard system tools - systemd-networkd, NetworkManager, etc.

Enabling streams (writing to /sys/class/forward/fd2110-*/io_config) - can be configured with a script, software (fopen(), fwrite() or libudev), or the udev rule

```
SUBSYSTEM=="forward", ATTR{software_id}=="<number>", ATTR{io_config}:="III----------O----------------------------------------------"
```

Example of confiration via libudev - `forward-v4l2-samples/common/Board.cpp`, function `configureIOMode()`

PTP setup

The standard Linux client, linuxptp, is used for PTP. Launch example:

```
ptp4l -i enp1s0d1 -H -s -m
```

After that, the board will be synchronous to PTP.

V4L2/ALSA input setup

The stream is configured for reception via the V4L2 and ALSA controls. It is set using controls:

- Video timings - 1080i50, 1080p50, etc.
- Video stream type - ST2110-20 or ST2022-6
- The address (IP + Port) from where the video stream will be received.
- The color format of the stream (8/10-bit, YCbCr/RGB). So far, only YCbCr, 8 or 10 bits are supported.
- The address (IP + Port) from where the audio stream will be received.

An example of setting parameters via the V4L2 API and video capture is given in `forward-v4l2-samples/capture-fd2110/`

All this can be done using a shell script, `v4l2-ctl` or `qv4l2`:

```
# Enable first input and first output
echo III---------------O---------------------------------------------- > /sys/class/forward/fd2110-*/io_config

# Use 1080p50 timings on input
v4l2-ctl -d /dev/video2 --set-dv-bt-timings index=13

# Setup input stream type (0 - ST2110, 1 - ST2022-6)
v4l2-ctl -d /dev/video2 -c stream_type=0
# Setup input address
v4l2-ctl -d /dev/video2 -c stream_address='239.21.10.1:10000'
# YCbCr 4:2:2
v4l2-ctl -d /dev/video2 -c stream_color_mode=0
# 10-bit
v4l2-ctl -d /dev/video2 -c stream_bits_per_component=1

# ALSA setup
amixer -c 0 cset "iface=PCM,name='PCM Capture Stream
```

```
Type',device=2,subdevice=1" 0x00
amixer -c 0 cset "iface=PCM,name='PCM Capture Stream
Address',device=2,subdevice=1"
"0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,0xff,0xef,0x15,0x0a,0x01


# Enable multicast receiving on network interface
ip a add 239.21.10.1 dev enp1s0d1 autojoin
```

After that, the devices start working like regular video and audio devices - you can use `gstreamer` or `qv4l2` for tests

```
gst-launch-1.0 v4l2src device=/dev/video2 ! autovideosink
```

V4L2/ALSA output setup

The stream is also configured for transmission via the V4L2 and ALSA controls. It is set using controls:

- Video timings - 1080i50, 1080p50, etc.
- The address (IP + Port) to where the video stream will be sent.
- The color format of the stream (8/10-bit, YCbCr/RGB). So far, only YCbCr, 8 or 10 bits are supported.
- Outgoing port number
- Payload Type for RTP (usually 96)
- The address (IP + Port) to where the audio stream will be sent.

All this can be done using a shell script, `v4l2-ctl` or `qv4l2`:

```
# Enable first input and first output
echo III---------------O---------------------------------------------- >
/sys/class/forward/fd2110-*/io_config

# Use 1080p50 timings on output
v4l2-ctl -d /dev/video3 --set-dv-bt-timings index=13

# Setup output address
v4l2-ctl -d /dev/video3 -c stream_address='239.21.10.2:10000'
# YCbCr 4:2:2
v4l2-ctl -d /dev/video3 -c stream_color_mode=0
# 10-bit
v4l2-ctl -d /dev/video3 -c stream_bits_per_component=1
# Outgoing port
v4l2-ctl -d /dev/video3 -c stream_source_port=10000
# Payload type
v4l2-ctl -d /dev/video3 -c stream_payload_type=96

# ALSA setup
amixer -c 0 cset "iface=PCM,name='PCM Playback Stream Source
Port',device=18,subdevice=0" 10001
amixer -c 0 cset "iface=PCM,name='PCM Playback Stream
Address',device=18,subdevice=0"
"0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,0xff,0xef,0x15,0x0a,0x02
```

After that, the devices start working like regular video and audio devices - you can use `gstreamer` or `qv4l2` for tests

```
gst-launch-1.0 videotestsrc ! v4l2sink device=/dev/video3
```

Seamless Protection Switching (SPS, ST2022-7) on output

To duplicate the output stream, you need to configure the stream addresses on both ports, and then set the control st2022_7_tx_enable

```
v4l2-ctl -d /dev/video3 -c st2022_7_tx_enable=1
```

After that, when opening on the first port, the data will be duplicated on the second, while the addresses on the second port will be replaced. Devices on the second port will not work.

Full example:

```
# Enable streams on both ports
echo III---------------O---------------I---------------O--------------- >
/sys/class/forward/fd2110-*/io_config
# First port broadcast to 239.21.10.2:10000 and 239.21.10.2:10001
v4l2-ctl -d /dev/video3 -c stream_address="239.21.10.2:10000"
amixer -c 1 cset "iface=PCM,name='PCM Playback Stream Source
Port',device=18,subdevice=0" 10001
amixer -c 1 cset "iface=PCM,name='PCM Playback Stream
Address',device=18,subdevice=0"
"0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,0xff,0xef,0x15,0x0a,0x02

# Second port broadcast to 239.21.10.4:10000 and 239.21.10.4:10001
v4l2-ctl -d /dev/video5 -c stream_address="239.21.10.4:10000"
amixer -c 1 cset "iface=PCM,name='PCM Playback Stream Source
Port',device=50,subdevice=0" 10001
amixer -c 1 cset "iface=PCM,name='PCM Playback Stream
Address',device=50,subdevice=0"
"0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xff,0xff,0xef,0x15,0x0a,0x04

# Enable SPS
v4l2-ctl -d /dev/video3 -c st2022_7_tx_enable=1
```

Changelog

Date	Driver version	Changes
2026.03.19	1.7.13	ST2022-6 support
2026.01.22	1.7.7	Typo fixes and english version
2025.12.12	1.7.7	Raw-ASI examples
2025.12.05	1.7.6	Docs for ASI, Raw, Complex
2025.12.03	1.7.6	Initial version